I l6N
no. 852
Cop 2

UIUCDCS-R-77-852

UILU-ENG 77 1710

A Formal Definition of the SIBYL Programming Languages

by

Garry Kampen

March 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

A Formal Definition of the SIBYL Programming Languages

by

Garry Kampen

Department of Computer Science
University of Illinois
Urbana, Illinois

March 1977

NEW TITLE

THIRD PANEL

FOURTH PANEL

CALL NUMBER

CHANGE CALL NO

ADD'N. T CALL NO

NO TRIM

MAP POCKET

COLOR NUMBER

2. INSERT IN THE VOLUME

23

Abstract

TITLE NO.	ACCOUNT NO.	LOT AND TICKET NO.
2-6263.5000	07200-009	97-00053

2-6 ILLINOIS * UNIVERSITY * DEPT. OF * COMPUTER *
SCIENCE * 43-12 REPORT *

3-24 1977 * NO.852-857 *

2-18 510.84*I2L6aR*
NO.852-857*COP.2* MTH 2. WATCH MARGINS.

CLOTH COLOR 0091

HEIGHT 11 0/0

CHARGING INFORMATION						HEIGHT	PICA	WRAP
STUBBING			CALL NO.					
HINGING			EXTRA LETTERING					
EXTRA TIME			BINDING CHARGES					
EXTRA THICK								
HAND SEW								

tion of
ves
ctures
YL is
an
a sequence
ics
r and
used
other
ed for
s is

TITLE NO.	ACCOUNT NO.	LOT AND TICKET NO.
2-6263.5000	07200-009	97-00053

42-6 ILLINOIS * UNIVERSITY * DEPT. OF * COMPUTER *
SCIENCE * 43-12 REPORT *

v10

43-24 1977 * NO.852-857 *

42-18 510.84*I2L6aR*
NO.852-857*COP.2* MTH 2. WATCH MARGINS.

CLOTH COLOR 0091

HEIGHT 11 0/0

CHARGING INFORMATION						HEIGHT	PICA	WRAP
STUBBING			CALL NO.			11¼	8	T
HINGING			EXTRA LETTERING					
			BINDING CHARGES					

EXTRA TIME	
EXTRA THICK	
HAND SEW	

Abstract

This report contains a complete formal description of an experimental programming language, SIBYL, which achieves simplicity by generalizing a number of concepts and structures found in other programming languages. The syntax of SIBYL is defined by a context free grammar, and the semantics by an interpreter whose state transition function consists of a sequence of string transformation rules. This approach to semantics provides an operational definition that is highly modular and well adapted to mechanical verification.

Because of its generality, SIBYL can in turn be used as a language definition tool. By mapping constructs in other high-level languages into their SIBYL equivalents, the need for repeated definitions of common control and data structures is avoided.

A Formal Definition of the SIBYL
Programming Language

Table of Contents

List of Figures

A Formal Definition of the SIBYL Programming Language

1.0 Introduction.

The design and implementation of a software system requires a variety of artificial languages: job control languages, programming languages, notations for system design and analysis and metanotations to describe the other languages. The diversity of these languages makes life more difficult for the programmer who must learn and use them, but it appears to be inevitable. Existing languages are retained because they possess powerful compilers, extensive subroutine libraries, and numbers of experienced users. New languages are developed because they are better suited to specialized problem areas or user groups, or in order to take advantage of new hardware or software concepts.

In the face of this linguistic diversity, two trends offer hope to the beleaguered computer user: the gradual replacement of relatively specialized and idiosyncratic languages like Fortran and COBOL by more general and powerful languages like PL/1 and Algol 68, and the development of metanotations for the precise specification of programming languages. This paper describes a syntactic and semantic metanotation aimed specifically at the users of programming languages. The formalism supports language descriptions that are understandable by mathematically unsophisticated programmers, and sufficiently complete and precise to be transformed mechanically into compilers or interpreters.

An extended example of the metanotation is provided in sections 4 and 5, which contain the formal syntactic and semantic specification of a self-extensible interactive block-structured typeless language, SIBYL. SIBYL is a high-level programming language that incorporates features from languages as diverse as APL, LISP, ALGOL 68, PASCAL, and GEDANKEN. Data structures include integer and decimal numbers, Boolean values, character strings, pointers, lists and records. Arrays, files, stacks, and queues are provided as special cases of lists. Control structures include recursive procedures, a variety of loops, and extended case and conditional expressions. Scalar operators extend to arrays as in APL. For an example program, see figure 1.

A unique feature of SIBYL is the provision of explicit operators for accessing and modifying the local environment of blocks and procedures. This permits extremely flexible scope rules. For example, a block or procedure can be given access to any or none of the following: variables in enclosing blocks, selected common or own variables, the environment of definition of the procedure, the environment of execution of the procedure.

SIBYL is designed to serve in two distinct but complementary roles: the intermediate language of a two-level language definition system, and a high-level problem description language in a programming system. The first role is essentially that of a common base language as described by Dennis [Dennis, 1972].

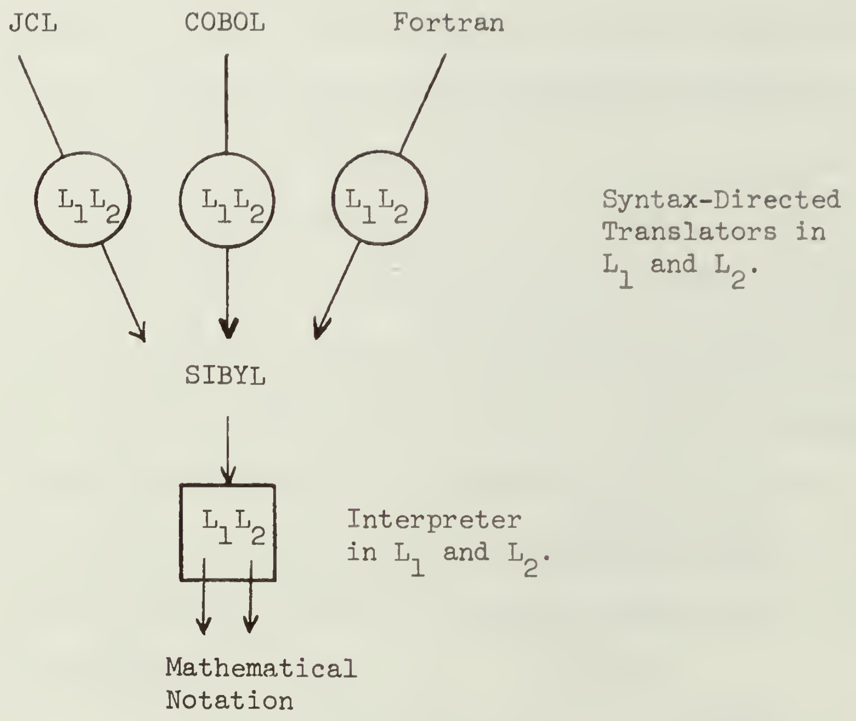
The two-level approach to language definition is illustrated in figure 2. Programming language semantics are defined by syntax-directed translators which map source language constructs into equivalent SIBYL constructs. The semantics of SIBYL are defined by

```
Job(id:12345,time:20,line:500);
(in:Data(device:Database,...)
,out:Data(device:Sysprint,...)
,Main:[
      C'Part cost summary.'
      (input:ref(in),
       output:ref(out),
       part:(name:'xxxxxxx',
             cost:999     ),
       sum:0              )$
      (len(input)@
         read(input) → part;
         part.cost + sum → sum );
         'OTotal is ' sum →> output
])$
!Main;
```

The SIBYL code above reflects in abbreviated form the structure of a COBOL program and its accompanying JCL. The 'execute program' command !Main has exactly the same effect as the procedure call Query(in,out) when Query is defined by

```
Query:[(i,o)$('OTotal is '(i@cost)+)→>o].
```

Figure 1: SIBYL Example.



JCL
COBOL
Fortran
L_1L_2
L_1L_2
L_1L_2
Syntax-Directed Translators in L_1 and L_2.
SIBYL
L_1L_2
Interpreter in L_1 and L_2.
Mathematical Notation

Figure 2: Language Definition System.

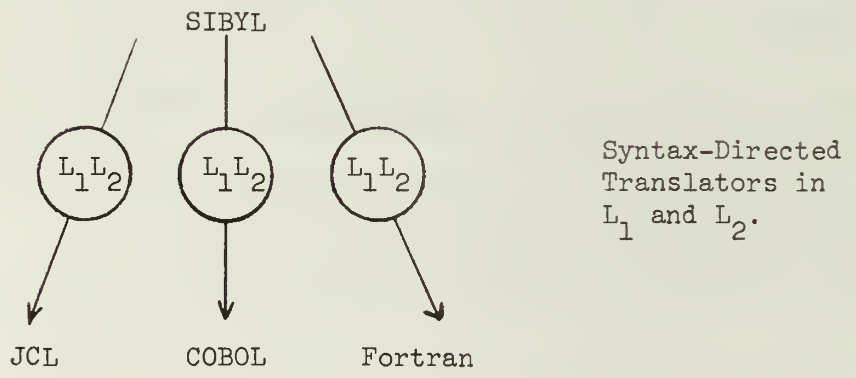


Figure 3: Programming System.

an interpreter. The translators and interpreter are described by transition rules from the semantic metalanguage L_2, which operate on syntactic constructs defined in the metalanguage L_1. L_1 and L_2 are simple enough to be informally defined, or to be given a straightforward mathematical description.

An advantage of the two-level approach is that both the translators and the interpreter can be relatively compact: the translators, because SIBYL is 'close' to the source language in the sense that it includes many high-level semantic constructs; the interpreter, because SIBYL is an almost purely <u>expressive</u> language. <u>Repressive</u> features, that is, features which restrict the generality of the language in order to permit efficient compilation or execution, are deliberately omitted.

The use of SIBYL in a programming system is illustrated in figure 3. A description of the problem and its initial solution program are written in SIBYL and checked for correctness. When this has been done, critical procedures are reprogrammed for greater efficiency and translated into languages for which efficient compilers exist. This approach reduces a large problem to a pair of smaller ones: Finding a correct description of the solution, and obtaining an efficient program. Moreover, SIBYL's freedom from hardware-oriented features ensures a high degree of portability for the initial program.

The syntactic and semantic metalanguages L_1 and L_2 are defined informally in sections 3.1 and 3.2 respectively. The elements of L_1

are sets of context-free syntax productions; L_2 contains lists of state transition rules which operate on strings of characters.

Transition rules can be grouped into semantic modules which define separate language components, thus permitting a building-block approach to language design. Notation for describing the behavior of these building-blocks is introduced in sections 3.3 and 3.4.

Operators for combining semantic modules into networks of interacting automata are introduced in sections 3.5 and 3.6. The resulting metanotation L_3 can be used to define the hardware components of a programming system.

The metanotations L_1 and L_2 are used in sections 4 and 5 respectively to define the syntax and semantics of SIBYL. L_3 is used in section 6 to describe a multiprocessor environment in which SIBYL programs can initiate parallel tasks.

2.0 Background.

Before discussing the SIBYL metanotation in more detail, it will be useful to review existing syntactic and semantic formalisms. For syntax, the most widely used notations are variations of (a) the two-dimensional metalanguage used to define COBOL [US, 1965], and (b) the BNF notation introducted by Backus [Backus, 1959].

These notations are essentially equivalent, their meaning is well understood, and implementations exist in the form of parse-table generators which accept grammars as input [Aho, Ullman, 1972]. Other syntactic formalisms, notably two-level grammars [van Wijngaarden, 1975] and the abstract syntax of the Vienna group [Lucas, Lauer, Stigleitner, 1970], can best be viewed as components of the semantic notations they support.

Existing semantic methods exhibit varying degrees of formalism and abstractness, ranging from the specification of a language by its compiler [Garwick, 1966] to a purely axiomatic description [Hoare, 1969] or an algebraic characterization [Goguen, Thatcher, Wagner, Wright, 1975]. The methods lying between these extremes can be loosely grouped into three categories: interpretive, devolutional, and functional.

Interpretive methods define a language by exhibiting an interpreter that transforms the current state of the computation into its successor state [Wegner, 1971]. The state includes a program and a memory component, and possibly separate control and environment components as well [Herriot, 1971]. The program component may be represented by a character string corresponding to the concrete program being executed [Kampen, 1973] or by an abstract object resembling a parse tree, as in the Vienna group method. State transitions may be described by functions [Lucas, Walk, 1969], by precise but informal descriptions [van Wijngaarden, 1975], or by programs in a simple language which is assumed to be already known [Lukaszewicz, 1976].

Devolutional descriptions provide a translation algorithm which maps programs in the language being defined into programs in a language which is assumed to be already known [Wirth, Weber, 1969]. Devolutional definitions may be wholly or partly extensional: a language is mapped into a subset of itself either by an external transformation rule or a statement within the language itself [Irons, 1970].

Functional and axiomatic methods tend to be implicit rather than constructive. For example, in the axiomatic system of Hoare [1969] the properties of a successor state are described but no method is given for computing it. Other approaches define a language by mapping programs into functions [Scott, Strachy, 1971; Tennant, 1976] or lambda expressions [Landin, 1965], or by functions which map programs into computations or terminal states [Hoare, Lauer, 1973].

A recent comparison of four major approaches to semantic definition [Marcotty, Ledgard, Bochmann, 1976] illustrates some of the drawbacks of existing formalisms: They are frequently harder to learn and use than the language they define; incapable of providing a clear, concise, easily modified description; and, with the exception of a partial implementation of the Vienna Definition Language [Feyock, 1975], largely unsupported by mechanical aids for verifying that descriptions are well-formed and logically consistent.

The design of the SIBYL metanotation has been guided by the desire to provide a formalism that is

(a) sufficiently simple to be readily learned by programmers, and

(b) sufficiently complete so that, given a language description, sample programs and their computations can be mechanically generated.

In order to achieve simplicity, standard syntactic constructs are shared by SIBYL and its metanotation wherever possible. For example, names and character strings have a standard representation. Semantically, L_1 is an extended BNF; L_2 resembles (but is not identical to) a subset of the well-known string-manipulation language SNOBOL.

To permit mechanization, definitions are interpretive rather than axiomatic. Statements equivalent to axioms in the Hoare notation must be proved as theorems. Programs and computational states are represented by concrete character strings. Abstractions arise from and guide the definition but are not part of it.

Considerations of simplicity have guided the <u>use</u> of the metanotation as well as its design. The grammar for SIBYL has been simplified by minimizing the amount of semantic information it conveys: operator procedence, for example, is only suggested by the grouping of operators into classes. The SIBYL interpreter is defined by a list of approximately 100 semantic rules, none involving recursion. A definition in terms of recursive functions might have reflected the structure of the language more clearly, but it would not have permitted the incremental approach of section 5, where the language is developed as a series of successively more complex sublanguages.

3.0 Notation and Terminology.

3.1 Syntactic Metalanguage.

The syntactic metalanguage L_1 is a set of grammars, each consisting of one or more productions of the form

$$n \rightarrow e$$

where n is the name of a syntactic class (non-terminal symbol) and e is an expression whose operands are syntactic class names and strings of characters (terminal symbols) over an alphabet V. Strings are enclosed in quotes or underlined; syntactic class names begin with an upper-case letter. The operators |, +, and * mean 'or', 'one or more', and 'zero or more' respectively. For example, a class of integer expressions is defined by

```
Expression → Operand (Operator Operand)*
     Operator → '+'|'*'
     Operand → Nil|Digit+
        Nil → 0
        Digit → 0|1|2|3|4|5|6|7|8|9
```

Operator | has the lowest precedence, + and * the highest. For readability, we adopt the convention of indicating the components of a syntactic class by indentation.

L_1 includes expressions which define n-tuples of syntactic classes. For example, the class of states of a programmable pocket calculator with stack, display, and input registers and function keys + (add), * (multiply), $\vdash$ (clear) and $\dashv$ (evaluate) is given by the grammar

State $\rightarrow$ (Stack, Display, Input)

Stack $\rightarrow$ $\underline{\vdash}$ | $\underline{\vdash}$ Operand Operator

Display $\rightarrow$ Operand

Input $\rightarrow$ Key*

Key $\rightarrow$ Digit | $\underline{+}$ | $\underline{*}$ | $\underline{\vdash}$ | $\underline{\dashv}$

3.2 <u>Semantic Metalanguage</u>.

The semantic metalanguage L_2 is a set of <u>semantic descriptions</u> consisting of one or more <u>transition rules</u> of the form

$$(p_1, p_2, \ldots, p_n) \rightarrow (e_1, \ldots, e_m)$$

where the p_i are <u>patterns</u>, i.e. sequences of character strings and string variables, and the e_i are <u>string expressions</u>, i.e. sequences of character strings, string variables, and string-valued functions of string expressions. Variables appearing on the right side of a rule must appear on the left side of the same rule. For example, a possible semantic description D for the pocket calculator of section 3.1 is

$$\begin{array}{llcl}
D_1: & (s, w, y\,\underline{\vdash}\,y1) & \rightarrow & (\underline{\vdash}, \text{nil}, y1) \\
D_2: & (s, w, \text{digit}\ y) & \rightarrow & (s, w\ \text{digit}, y) \\
D_3: & (\underline{\vdash}, w, \text{op}\ y) & \rightarrow & (\underline{\vdash} w\ \text{op}, \text{nil}, y) \\
D_4: & (\underline{\vdash} w \underline{+}, w1, \text{key}\ y) & \rightarrow & (\underline{\vdash}, \text{Sum}(w, w1), \text{key}\ y) \\
D_5: & (\underline{\vdash} w \underline{*}, w1, \text{key}\ y) & \rightarrow & (\underline{\vdash}, \text{Prod}(w, w1), \text{key}\ y)
\end{array}$$

where s ε Stack, w,w1 ε Operand, y,y1 ε Input, digit ε Digit, op ε Operator, key ε Key, and Sum and Prod return the sum and product respectively of integers represented as strings of digits. Since semantic descriptions have the power of Turing machines, Sum and Prod can themselves be defined by transition rules.

By convention, string variables are formed from lower-case letters and digits, while the names of functions and descriptions are capitalized.

An informal description of the pocket calculator can be written to match the transition rules: The clear key $\underline{\vdash}$ clears the stack and zeroes the display (D_1). As digits are entered they appear in the display (D_2). When the stack is empty a $\underline{+}$ or $\underline{*}$ keyin pushes the displayed operand and the operator onto the stack (D_3). When the stack is not empty a $\underline{+}$, $\underline{*}$, or $\underline{\dashv}$ keyin causes the stack and display operands to be added or multiplied, depending on the stack operator, and the result placed in the display (D_4,D_5).

The semantics of L_2 insure that every description D defines a function. Given an n-tuple (s_1, s_2, ...,s_n) of strings we compute its value under D as follows:

(1) For every transition rule in D, match the patterns p_i against the corresponding strings s_i. Pattern-matching is done by a top-down parse with backup, as in the SNOBOL language. At each stage, the alternative that matches the longest string is tried first, then the next-longest, and so on. Patterns are matched from left to right, starting with p_1. When a variable is matched with a substring, subsequent occurrences of that variable are replaced with that substring.

(2) Choose the <u>first</u> (topmost) rule whose left-hand side matches the n-tuple, and compute D(s_1, s_2, ..., s_n) by evaluating the string expressions on the right-hand side.

The way rules (1) and (2) operate to eliminate semantic ambiguity is illustrated by the state below, which can be parsed in three different ways.

```
       (⊢05+,06,⊣ ⊢9⊣ ⊢10⊣)
  D1:  (s    ,w ,y     ⊢yl )    Applied.
  D1:  (s    ,w ,y -yl     )    Eliminated by (1).
  D4:  (⊢w+  ,wl,key y     )    Eliminated by (2).
```

3.3 *String Automata.*

The function defined by the semantic description D is a member of a class of abstract machines call string automata. A *string automaton of n registers* is a mapping $T: A \rightarrow B$ where A and B are subsets of a set S of *states* consisting of n-tuples of character strings over an alphabet V. When T is a function the automaton is said to be *deterministic*; otherwise it is *non-deterministic*.

A *computation* is a sequence $s_0, s_1, \ldots$ of states such that $s_{i+1} = T(s_i)$. States in $S - A$ are called *halt states*. If a computation is finite and its last state s_k is a halt state, the computation is said to *terminate*, and s_k is its *final state*. Also, s_0 is the *initial state* of the computation, s_{i+1} is the *successor state* of s_i, and s_j is a *consequence* of s_i whenever $j \geq i$. These relationships may be stated more compactly as

$$s_o \overset{\cdot}{\rightarrow} s_k$$

$$s_i \rightarrow s_{i+1}$$

$$s_i \overset{*}{\rightarrow} s_j$$

where $\overset{\cdot}{\rightarrow}$, $\rightarrow$, and $\overset{*}{\rightarrow}$ are relations on S determined by T. When T is deterministic, $\overset{\cdot}{\rightarrow}$ and $\rightarrow$ are functions.

An example of a computation determined by D is given below. Following each state is the transition rule D_i which applies to that state.

$(s, w, y\underline{\vdash 5+6\dashv})$	D_1: Clear stack and display.
$(\underline{\vdash}, \underline{0}, \underline{5+6\dashv})$	D_2: Enter digit into display.
$(\underline{\vdash}, \underline{05}, \underline{+6\dashv})$	D_3: Push operator onto stack.
$(\underline{\vdash 05+}, \underline{0}, \underline{6\dashv})$	D_2: Enter digit into display.
$(\vdash 05+, \underline{06}, \underline{\dashv})$	D_4: Compute sum.
$(\underline{\vdash}, \underline{11}, \underline{\dashv})$	

3.4 Relations on Expressions.

String automata can be used to define software or hardware modules (e.g. pocket calculators) or languages (e.g. arithmetic expressions). In the latter case, it is convenient to extend the relations $\dot{\rightarrow}$, $\rightarrow$, and $\overset{*}{\rightarrow}$ from states to expressions. Assume that T is the defining automaton for a language L, and that s& is the string formed by concatenating the components of a state s, i.e.

$$(s_1, s_2, \ldots, s_n)\& = s_1 s_2 \ldots s_n.$$

Then the relation $\dot{\rightarrow}$ is defined for $e_1, e_2 \in L$ by

el $\dot{\rightarrow}$ e2 iff s1 $\dot{\rightarrow}$ s2 where s1 and s2 are states,

s1&= m1$\underline{\vdash}$e1$\underline{\dashv}$ for some string m1, and

s2&= m2$\underline{\vdash}$e2$\underline{\dashv}$ for some string m2.

Relations $\rightarrow$ and $\overset{*}{\rightarrow}$ are defined similarly. An equivalence relation $\equiv$ on expressions is defined by

$$e1 \equiv e2 \text{ iff } e1 \overset{*}{\rightarrow} e \text{ and } e2 \overset{*}{\rightarrow} e \text{ for some expression } e.$$

Enclosing quotes and underlining may be omitted when the meaning is clear. For example, when L is the class Expression defined in 3.1 and T is defined by D from section 3.2, then

$$5+6 \overset{*}{\rightarrow} 05+06 \rightarrow 11 \text{ and}$$
$$5+6 \equiv 6+5.$$

3.5 Operations on Automata.

String automata may be combined to form new automata by a variety of operators, including the function composition operator o:

$$T = T_1 \circ T_2 \quad \text{iff} \quad T(s) = T_1(T_2(s)) \quad \text{for all } s.$$

$$T = T_1 \mathbin{\&} T_2 \quad \text{iff} \quad T(s) = T_1(s) \text{ when } T_1(s) \text{ is defined,}$$
$$= T_2(s) \text{ otherwise.}$$

$$T = T_1{}^* \quad \text{iff} \quad T(s1) = s2 \text{ when } s1 \rightarrow s2 \text{ in } T_1.$$

$$T = T_1{}^n \quad \text{iff} \quad T = T_1 \text{ when } n = 1,$$
$$= T_1 \circ T_1{}^{n-1} \text{ otherwise.}$$

For example, the expression below defines a new pocket calculator which behaves like the original except that operands are entered all at once instead of a digit at a time.

$$New = D_1 \ \& \ D_2^* \ \& \ D_3 \ \& \ D_4 \ \& \ D_5$$

An abbreviated form of the same expression is

$$New = D_1 \ \& \ D_2^* \ \& \ D_{3..5}$$

3.6 Networks of Automata.

If T is a string automaton with states of the form $s = (s_1, s_2, \ldots s_n)$, and if $R = (R_1, R_2, \ldots R_n)$ is a sequence of n distinct objects called <u>registers</u>, then T(R) denotes an <u>instance</u> of T. The terminology introduced for string automata also applies to instances of string automata. When T(R) is in state s, string s_i is called the <u>content</u> of register R_i. A register can have only one content.

A <u>network</u> is a set of instances of string automata whose registers are selected from a list R. The state of the network is the list of contents of R. When a network component changes state, so does the network.

If $T_1(P)$ and $T_2(Q)$ are instances of the string automata

T_1 and T_2 and R is a list of the registers appearing in P and Q, the expression

$$T(R) = T_1(P) + T_2(Q)$$

defines T to be the string automaton whose successor state T(s) is defined as follows: Let s determine the contents of R, and apply T_2 to the contents of P. Apply T_1 to the state induced by the new contents of P. The resulting network state is T(s).

The module connection operator + can be defined more precisely in terms of the functions $T_1(R:P)$ and $T_2(R:Q)$ induced on states of $T(R)$ by T_1 and T_2:

$$T(R:Q)(s) = s' \text{ iff } T(q) = q' \text{ where}$$
$$s'_i = q'_j \text{ and } s_i = q_j \text{ when } R_i = Q_j,$$
$$s'_i = s_i \text{ otherwise.}$$

$$T(R) = T_1(P) + T_2(Q) \text{ iff } T = (T_1(R:P) \circ T_2(R:Q)) \;\&\; T_2(R:Q) \;\&\; T_1(R:P)$$

The operators o, &, and + are all associative, so an expression of the form

$$T = T_1 + T_2 + \ldots + T_n$$

is well-defined. A state transition of T is computed by applying T_n to the current state if possible, then applying T_{n-1} if possible, and so on. If none of the T_i modifies the current state, T is undefined for that state.

Figure 4 shows the network defined by

$$\text{Net}(R1, R2, R3, R4) = D(R1, R2, R3) + \text{Term}(R3, R4) \text{ with}$$

Term_1: $(y,\underline{\text{start}}\ e\ \underline{\text{stop}}) \to (y\underline{\vdash}e,\underline{\text{stop}})$

Term_2: $(\lambda,\ \underline{\text{stop}}) \to (\lambda\ ,\underline{\dashv}\ )$

Term_3: $(\lambda\ ,\ \underline{\dashv}) \to (\underline{\dashv}\ ,\lambda\ )$

where $e \in$ Expression. When connected to the input buffer of the calculator, the terminal device defined by Term ensures that only legal expressions are transmitted, and permits the use of start and stop as syntactic sugar.

A sample computation is given below. The computations of D and Term represent parallel processes that pause while awaiting input. Note that D changes state twice during one transition of Net.

Net:	D:	Term:
$(\underline{\vdash 05+},\underline{06},\lambda,\underline{\text{stop}})$	$(\underline{\vdash 05+},\underline{06},\lambda)$	$(\lambda,\underline{\text{stop}})$
$(\underline{\vdash 05+},\underline{06},\lambda,\underline{\dashv}\ )$		$(\lambda,\underline{\dashv}\ )$
	$(\underline{\vdash 05+},\underline{06},\underline{\dashv})$	$(\underline{\dashv},\lambda\ )$
$(\underline{\vdash}\ ,11,\underline{\dashv},\lambda\ )$	$(\underline{\vdash}\ ,11,\underline{\dashv})$	

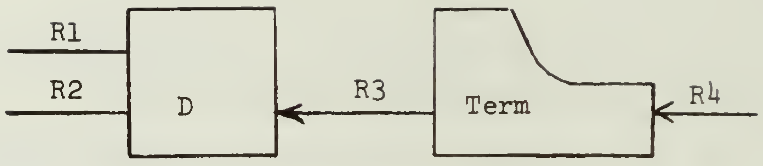


Figure 4: Network Example.

If A, B, and C are string automata, if R is a list of registers, and if we extend the operators &, o, *, and + by the definitions

$$A(R) \mathbin{\&} B(R) = (A \mathbin{\&} B)(R)$$
$$A(R) \circ B(R) = (A \circ B)(R)$$
$$A(R)^* = A^*(R)$$

then &, o, and + are associative, and moreover:

$$A \mathbin{\&} B = B \mathbin{\&} A \qquad \text{when } \mathrm{dom}(A) \cap \mathrm{dom}(B) = \emptyset$$
$$A \circ (B \mathbin{\&} C) = (A \circ B) \mathbin{\&} (A \circ C) \qquad \text{when } \mathrm{dom}(B) \cap \mathrm{dom}(C) = \emptyset$$
$$(A \mathbin{\&} B) \circ C = (A \circ C) \mathbin{\&} (B \circ C)$$
$$(A^*)^* = A^*$$
$$A + B = B \mathbin{\&} A \qquad \text{when } \mathrm{dom}(A) \cap \mathrm{rge}(B) = \emptyset$$
$$A + B = A \circ B \qquad \text{when } \mathrm{dom}(A) \subseteq \mathrm{rge}(B) \subseteq \mathrm{dom}(B)$$

4.0 SIBYL Syntax.

```
                State → (Memory, Stack Operand, Input)
m:                Memory → Record
s:                Stack → (Operand Operator)*
y:                Input → (Separator|Operator|Term)*
sep:                Separator → Blank+|Comment
op:                 Operator → Evaluator|Delimiter|(|,|)
term:               Term → Operand|Subexpression
w:                     Operand → Nil|Value
nil:                     Nil → Blank
v:                       Value → Constant|Variable
con:                        Constant → Primitive|Structure|Procedure
prim:                          Primitive → Null|Boolean|Token|Number|String
stru:                          Structure → List|Record
proc:                          Procedure → [Form]
                       Subexpression → (Form)
f:                        Form → (Statement|,)*
t:                           Statement → (Expression|Delimiter)*
e:                             Expression → (Separator|Evaluator|Term)*
```

The remainder of the context-free syntax for SIBYL is given below. The lower case name preceding each class name will be used to denote a string of that class. Names of distinct strings will be distinguished by an integer suffix. A string of the class (Operator Input) will be denoted by x. An example of a state of the form (m, s w op w1,x) is

$$(\underline{(A:5,B:6,)} \ , \underline{\vdash A+B} \quad ,\underline{\dashv})$$

4.1 Operators.

ev: $\text{Evaluator} \rightarrow \text{Op}_{10} \mid \text{Primary} \mid \text{Op}_6 \mid \text{Op}_2$

pr: $\text{Primary} \rightarrow \text{Op}_9 \mid \text{Op}_8 \mid \text{Op}_7 \mid \text{Relation} \mid \text{Op}_4 \mid \text{Op}_3$

rel: $\text{Relation} \rightarrow \text{Op}_5$

del: $\text{Delimiter} \rightarrow \text{Op}_1$

$\text{Op}_{10} \rightarrow \underline{!} \mid \underline{.!}$

$\text{Op}_9 \rightarrow \underline{*} \mid \underline{/} \mid \underline{\#}$

$\text{Op}_8 \rightarrow \underline{+} \mid \underline{-}$

$\text{Op}_7 \rightarrow \underline{..}$

$\text{Op}_6 \rightarrow \underline{\&} \mid \underline{.\&}$

$\text{Op}_5 \rightarrow \underline{<} \mid \underline{>} \mid \underline{=} \mid \underline{<=} \mid \underline{>=} \mid \underline{/=}$

$\text{Op}_4 \rightarrow \underline{\neg} \mid \underline{\wedge}$

$\text{Op}_3 \rightarrow \underline{\vee}$

$\text{Op}_2 \rightarrow \underline{\rightarrow} \mid \underline{\rightarrow>} \mid \underline{.\rightarrow} \mid \underline{.\rightarrow>} \mid \underline{.}\text{Op}_1$

$\text{Op}_1 \rightarrow \underline{:} \mid \underline{;} \mid \underline{\$} \mid \underline{=>} \mid \underline{@} \mid \underline{\vdash} \mid \underline{\dashv}$

$\text{Op}_0 \rightarrow \underline{(} \mid \underline{,} \mid \underline{)}$

The operators above are grouped by precedence from highest (Op_{10}) to lowest (Op_0).

4.2 Primitive Values and Comments.

prim:	Primitive → Null|Boolean|Token|Number|String
null:	Null → %
bool:	Boolean → True|False
true:	True → 1
false:	False → 0
token:	Token → Function|Ptype|Stype|Deref
fun:	Function → hd|tl|len|dom|rge|read|pop
ptype:	Ptype → null|bool|token|num|int|dec|str
stype:	Stype → list|rec|proc
deref:	Deref → val|ref|con
num:	Number → Index|Integer|Decimal
i:	Index → Digit+
int:	Integer → Sign Digit+
dec:	Decimal → Sign Digit+ . Digit+
	Sign → λ|-
str:	String → Quote Characters Quote
	Quote → '
cs:	Characters → Character*
c:	Character → Letter|Digit|Blank|Quote Quote|Other
	Comment → C String

Some examples of primitives are given below. Note that the quote character is represented within a string by a pair of quotes.

```
% 	0 	token 	12 	-3 	12.5 	'O''CONNOR' C'comment.'
```

4.3 Data Structures and Variables.

```
list:  List → ( Values )

vs:      Values → ( Value , )*

rec:   Record → ( Fields )

fs:      Fields → (Field,)*
            Field → Name : Value
name:       Name → Letter (Letter|Digit)*

var:   Variable → Name|Name.Subscripts|Reference

ref:     Reference → r.Subscripts

subs:       Subscripts → Subscript (.Subscript)*

sub:            Subscript → Name|Index
```

A record r with a list and a record as values is exhibited below along with some variables that reference its components.

```
r:(ARRAY:((1,),(1,2,),(1,2,3,),),EMPLOYEE:(NAME:'JONES',AGE:30,),)

r.ARRAY   ARRAY   ARRAY.3.2   EMPLOYEE.NAME
```

4.4 Character Set.

```
sym:  Symbol → Letter|Digit|Blank|Quote|Other

        Letter → Uppercase|Lowercase

        Other → Sym1|Sym2|Sym3|Sym4|Nonascii|Control

           Blank → _

           Sym1 → !|"|#|$|%|&

           Quote → '

           Sym2 → (|)|*|+|,|-|.|/

           Digit → 0|1|2|3|4|5|6|7|8|9
```

Sym3 → $\underline{:}\,|\,\underline{;}\,|\,\underline{<}\,|\,\underline{=}\,|\,\underline{>}\,|\,\underline{?}\,|\,\underline{@}$

Uppercase → $\underline{A}\,|\,\underline{B}\,|\,\ldots\,|\,\underline{Z}$

Sym4 → $\underline{[}\,|\,\underline{\backslash}\,|\,\underline{]}$

Lowercase → $\underline{a}\,|\,\underline{b}\,|\,\ldots\,|\,\underline{z}$

Nonascii → $\underline{\vdash}\,|\,\underline{\dashv}\,|\,\underline{\neg}\,|\,\underline{\wedge}\,|\,\underline{\vee}$

Control → ASCII control characters

All symbols except $\underline{\vdash}$, $\underline{\dashv}$, $\underline{\wedge}$, $\underline{\vee}$, and $\underline{\neg}$ are members of the extended ASCII character set. The exceptions will be represented externally by the keywords start, stop, and, or, and not. Uppercase characters may be substituted for lowercase when the latter are not available.

ASCII symbols other than control characters are listed in the order they appear in the ASCII collating sequence.

5.0 SIBYL Semantics.

The formal semantic definition of SIBYL is presented in this section. The definition is a list of semantic rules which describe an interpreter with states of the form (m,s w,y).

Semantic rules are grouped and ordered to parallel the syntax productions of section 4, and to make possible a step-by-step development of the language: At each step, the rules to that point define a subset of the overall language, and examples are drawn from this sublanguage.

5.1 Expressions.

E_1: $(m,s\quad w\quad ,y\underline{\vdash}\quad y) \rightarrow (m,nil\ \underline{\vdash}\quad nil,\quad yl)$

E_2: $(m,s\quad w\quad ,sep\ y) \rightarrow (m,s\quad w\ ,\quad y)$

E_3: $(m,s\quad nil,v\quad y) \rightarrow (m,s\quad v\ ,\quad y)$

E_4: $(m,s\quad v\quad ,vl\quad y) \rightarrow (m,s\quad v\quad ,\underline{!}vl\ y)$

E_5: $(m,s\quad w\quad ,\underline{(}\quad y) \rightarrow (m,s\quad w\ \underline{(}\ nil,\quad y)$

E_6: $(m,s\ w\ \underline{(}\ v\quad ,\underline{)}\quad y) \rightarrow (m,s\quad w,v\ nil\ y)$

E_7 $(m,s\quad nil\quad ,ev\quad y) \rightarrow (m,s\ nil\ ev\ nil,\quad y)$

E_8 $(m,s\ w\ op_i\ wl,op_j\ \ y) \rightarrow (m,s\ w\ op_i\ wl\ op_j\ nil,\ y)$

where $0 \leq i < j$ and $1 < j$.

The operator $\underline{\vdash}$ clears and resets the stack and deletes everything to its left in the input buffer (rule E_1). Since the first rule takes precedence over the others, a $\underline{\vdash}$ keyin can be used to halt an otherwise nonterminating computation.

Expressions are evaluated from left to right. Separators are ignored (E_2), but the operator $\underline{!}$ is inserted between adjacent values (E_4). Values are entered when the top of the stack is empty (E_3), and pushed down when the operator to their right has higher precedence than the operator to their left (E_8). Prefix operators (e.g. operators whose left argument is nil) take precedence over the operator to their left (E_7).

5.2 Primitive Values.

5.2.1 Null.

R_1: (m,s null $\underline{!}$ v,x) $\rightarrow$ (m,s null,x)

A null value applied to a value v returns null.

5.2.2 Booleans.

R_2: (m,s nil$\underline{\neg}$bool ,x) $\rightarrow$ (m,s Not(bool) ,x)

R_3: (m,s bool$\underline{\wedge}$booll,x) $\rightarrow$ (m,s And(bool,booll),x)

R_4: (m,s bool$\underline{\vee}$booll,x) $\rightarrow$ (m,s Or(bool,booll),x)

The Boolean operators $\underline{\neg}$, $\underline{\wedge}$, and $\underline{\vee}$ are defined by the functions Not, And, and Or, which have the usual meaning for the values $\underline{1}$ (true) and $\underline{0}$ (false):

$$\neg 1 \vee 0 \wedge 1 \overset{*}{\rightarrow} 0 \vee 0 \overset{*}{\rightarrow} 0$$

5.2.3 Tokens.

R_5: (m,s con $\underline{=}$ stype,x) → (m,s Type(con,stype), x)

R_6: (m,s prim $\underline{=}$ ptype,x) → (m,s Type(prim,ptype),x)

The operator $\underline{=}$ (is) tests its left-hand argument for membership in the syntactic class designated by the token on the right. Type(x,y) = $\underline{1}$ (true) when x is in y, and $\underline{0}$ (false) otherwise.

Ex: 12 = num ∧ 12.0 = dec $\overset{*}{\rightarrow}$ $\underline{1}$

5.2.4 Relations on Numbers.

R_7: (m,s w rel wl,rell y) → (m,s w rel wl,$\underline{\wedge}$ wl rell y)

R_8: (m,s num$\underline{<}$ numl, x) → (m,s Less(num,numl) ,x)

R_9: (m,s num$\underline{>}$ numl, x) → (m,s Less(numl,num) ,x)

R_{10}: (m,s num$\underline{=}$ numl, x) → (m,s Equal(num,numl) ,x)

R_{11}: (m,s num$\underline{<=}$numl, x) → (m,s nil$\underline{\neg}$Less(numl,num) ,x)

R_{12}: (m,s num$\underline{>=}$numl, x) → (m,s nil$\underline{\neg}$Less(num,numl) ,x)

R_{13}: (m,s num$\underline{/=}$numl, x) → (m,s nil $\underline{\neg}$ Equal(num,numl),x)

Expressions of the form e1 < e2 < ... are abbreviations for expressions of the form e1 < e2 ∧ e2 < ... (R_7). The relational operators $\underline{<}$, $\underline{>}$, $\underline{=}$, $\underline{<=}$, $\underline{>=}$, and $\underline{/=}$ yield a Boolean value when applied to numeric operands.

Ex: $12 = 12 = 012.0 \overset{*}{\rightarrow} 1\wedge 1 \overset{*}{\rightarrow} 1$

5.2.5 Numbers.

R_{14}: (m,s num $\underline{*}$ num1,x) → (m,s Prod(num,num1),x)

R_{15}: (m,s num $\underline{/}$ num1,x) → (m,s Quot(num,num1),x)

R_{16}: (m,s num $\underline{+}$ num1,x) → (m,s Sum(num,num1),x)

R_{17}: (m,s num $\underline{-}$ num1,x) → (m,s Diff(num,num1),x)

R_{18}: (m,s nil op_8 num ,x) → (m,s $\underline{0}$ op_8 num x)

R_{19}: (m,s num $\underline{!}$ v ,x) → (m,s num,$\underline{+}$ v x)

The basic arithmetic operators (including unary + and -) are defined by the functions Prod, Quot, Sum, and Diff which return the product, quotient, sum, and difference of their arguments. When the operator $\underline{!}$ is coerced to $\underline{+}$ it is moved to the input buffer to allow for the possibility that the operator to the left has higher precedence than $\underline{+}$ (R_{19}).

Ex: $12-3*25 \overset{*}{\rightarrow} 12+-3*25 \overset{*}{\rightarrow} 12+-75$

5.2.6 Relations on Strings.

R_{20}: (m,s str rel str1,x) → (m,s nil$\underline{\#}$str,rel$\underline{\#}$str1 x)

R_{21}: (m,s nil $\underline{\#}$ str ,x) → (m,s Encode(str) ,x)

R_{22}: (m,s nil $\underline{\#}$ i ,x) → (m,s Decode(i) ,x)

The function Encode maps every string into the unique non-negative integer given by its representation in 7-bit ASCII. Decode is the inverse of Encode. Strings of the same length are ordered lexicographically by the ASCII collating sequence. Short strings precede (are less than) longer strings.

Ex: 'B' < 'AB' $\overset{*}{\rightarrow}$ 1

5.2.7 <u>Strings</u>.

R_{23}: $\texttt{(m,s\underline{'}cs\underline{'\&} \underline{'}csl\underline{'},x)} \rightarrow \texttt{(m,s \underline{'}cs csl\underline{'} ,x)}$

R_{24}: $\texttt{(m,s str \underline{\&} con ,x)} \rightarrow \texttt{(m,s str \underline{\&}Qt(con),x)}$

R_{25}: $\texttt{(m,s str \underline{!} v ,x)} \rightarrow \texttt{(m,s str,\underline{\&} v x)}$

R_{26}: $\texttt{(m,s \underline{hd!'}c cs\underline{'},x)} \rightarrow \texttt{(m,s \underline{'}c\underline{'} ,x)}$

R_{27}: $\texttt{(m,s \underline{tl!'}c cs',x)} \rightarrow \texttt{(m,s \underline{'}cs\underline{'} ,x)}$

R_{28}: $\texttt{(m,s \underline{len!} str ,x)} \rightarrow \texttt{(m,s Len(str) ,x)}$

R_{29}: $\texttt{(m,s nil\underline{!} str ,x)} \rightarrow \texttt{(m,s nil,Unq(str) x)}$

The basic string operators are <u>&</u> (concatenate) and the functions <u>hd</u>, <u>tl</u>, and <u>len</u> which return the head (first character), tail (all but the first character), and length (number of characters) of a string. The <u>!</u> operator coerces to <u>&</u> when preceded by a string (R_{25}). Constants other than strings are concatenated after being coerced by the function Qt, which first replaces every quote in the constant by a pair of quotes, and then encloses the result in single quotes. Unq is the inverse of Qt. Since the result of the unquote operation may be an expression rather than an operand, it is placed in the input buffer (R_{29}).

Ex_1: hd'ABC' & tl'DEF' $\overset{*}{\rightarrow}$ 'A'&'EF' $\overset{*}{\rightarrow}$ 'AEF'

Ex_2: 'N=' 12+3 $\overset{*}{\rightarrow}$ 'N='&15 $\overset{*}{\rightarrow}$ 'N=15'

Ex_3: len'MC''COY'*!'12*3' $\overset{*}{\rightarrow}$ 6*12*3 $\overset{*}{\rightarrow}$ 216

5.3 <u>Data Structures</u>.

5.3.1 <u>Lists</u>.

D_1: $(m,s\ \underline{(}\ vs\ \underline{)\&}\ \underline{(}\ vsl\ \underline{)}\ ,x) \rightarrow (m,s\ \underline{(}\ vs\ vsl\ \underline{)}\ ,\ x)$

D_2: $(m,s\ \ \underline{\text{hd!}}\ \underline{(}\ v\underline{,}vs\ \underline{)},x) \rightarrow (m,s\ \ v\ \ ,\ x)$

D_3: $(m,s\ \ \underline{\text{tl!}}\ \underline{(}\ v,vs\ \underline{)},x) \rightarrow (m,s\ \underline{(}\ vs\ \underline{)}\ \ ,\ x)$

D_4: $(m,s\ \ \underline{\text{len!}}\ \ \text{list}\ \ ,x) \rightarrow (m,s\ \text{Len(list)}\ \ ,\ x)$

D_5: $(m,s\ \ \text{int}\ \underline{..}\ \text{intl}\ \ ,x) \rightarrow (m,s\ \text{Seq(int,intl)},\ x)$

The basic list operators are <u>&</u> (concatenation), <u>hd</u> (head), <u>tl</u> (tail), and <u>len</u> (length). As in LISP, <u>hd</u> and <u>tl</u> are undefined for the empty list <u>()</u>. The operator <u>..</u> (sequence) generates a list of consecutive integers in ascending or descending order starting with int and ending with intl.

Ex_1: 0..-2&0..2 $\overset{*}{\rightarrow}$ (0,-1,-2,)&(0,1,2,) $\overset{*}{\rightarrow}$ (0,-1,-2,0,1,2,)

Ex_2: hd(5,6)+len(tl(5,8)) $\overset{*}{\rightarrow}$ 5+len((8,)) $\overset{*}{\rightarrow}$ 6

5.3.2 List Subexpressions.

```
D6:  (m,s w ( vs nil,, x) → (m,s w ( vs     nil, x)

D7:  (m,s w ( vs v   ,, x)   (m,s w ( vs v,  nil, x)

D8:  (m,s w ( vs nil,) x)   (m,s w ,      (vs   ) x)

D9:  (m,s w ( vs v   ,) x)   (m,s w ,      (vs v,) x)
```

Subexpressions of the form (e_1, e_2, ..., e_n,) evaluate to lists of the form (v_1, v_2, ..., v_n,) where v_i is the value of expression e_i. Nil entries are deleted (D_6) and the abbreviation (e_1, e_2, ..., e_n) is permitted for lists of more than one element (D_9). Since (e_1) yields v_1 by a previous rule, it cannot evaluate to the singleton list (v_1,).

Ex: (5+6,,,7+8) $\overset{*}{\rightarrow}$ (11,15,)

5.3.3 Distribution.

```
D10:  (m,s list pr listl,x)  →  (m,s,Dist(list pr listl)x)

D11:  (m,s list pr prim ,x)  →  (m,s,Dist(list pr prim) x)

D12:  (m,s prim pr list ,x)  →  (m,s,Dist(prim pr list) x)
```

D_{13}: (m,s nil pr list ,x) → (m,s nil ,Dist(nil pr list) x)

D_{14}: (m,s con _.op conl ,x) → (m,s nil ,Dist(con op conl) x)

D_{15}: (m,s nil _.op con ,x) → (m,s nil ,Dist(nil op con) x)

Primary operators are applied componentwise to lists by the function Dist; the resulting list expression is placed in the input buffer and subsequently evaluated. For example

```
(5,6,) + (7,8,) → (5+7,6+8,) *→ (12,14,)
(5,6,) + 3      → (5+3,6+3,) *→ (8 , 9,)
3  < (5,6,)     → (3<5,3<6,) *→ (1 , 1,)
  - (5,6,)      → (-5 , -6,)
 len.!((),(),)  →  (len!(),len!(),)  *→  (0,0,)
```

The last two rules permit operators which take lists as arguments, like _!, to be distributed over lists. Distribution is defined by the string-valued function Dist = Dt*, where Dt is defined as follows:

$$\begin{array}{lllll}
\text{Dt:} & \text{f} & \underline{(}\underline{)}\ \text{op w} & \rightarrow & \underline{(}\text{f}\underline{)} \\
 & \text{f} & \text{w op}\ \underline{(}\underline{)} & \rightarrow & \underline{(}\text{f}\underline{)} \\
 & \text{f} & \underline{(}\text{t1}\underline{,}\text{f1}\underline{)}\ \text{op}\ \underline{(}\text{t2}\underline{,}\text{f2}\underline{)} & \rightarrow & \text{f t1 op t2}\underline{,}\underline{(}\text{f1}\underline{)}\ \text{op}\ \underline{(}\text{f2}\underline{)} \\
 & \text{f} & \underline{(}\text{t1 f1}\underline{)}\ \text{op}\ \underline{(}\text{t2 f2}\underline{)} & \rightarrow & \underline{(}\text{f t1 op t2}\underline{)} \\
 & \text{f} & \text{w op}\ \underline{(}\text{t2}\underline{,}\text{f2}\underline{)} & \rightarrow & \text{f w op t2}\underline{,}\ \text{w op}\ \underline{(}\text{f2}\underline{)} \\
 & \text{f} & \text{w op}\ \underline{(}\text{t2}\underline{)} & \rightarrow & \underline{(}\text{f w op t2}\underline{)} \\
 & \text{f} & \underline{(}\text{t1}\underline{,}\text{f1}\underline{)}\ \text{op w} & \rightarrow & \text{f t1 op w}\underline{,}\ \underline{(}\text{f1}\underline{)}\ \text{op w} \\
 & \text{f} & \underline{(}\text{t1}\underline{)}\ \text{op w} & \rightarrow & \underline{(}\text{f t1 op w}\underline{)} \\
 & \text{f} & \text{w op w1} & \rightarrow & \underline{(}\text{f w op w1}\underline{)}
\end{array}$$

Dt* is defined for all binary expressions whose terms are (a) a pair of subexpressions (t_1, t_2, ..., t_n) and (t_1',t_2', ..., t_m') or (b) a subexpression and an operand w. In the first case, the result is a subexpression of the form (t_1 op t_1', ..., t_k op t_k'), where k is the minimum of n and m. In the second case, the result is a subexpression of the form (t_1 op w, ..., t_n op w) or (w op t_1, ..., w op t_n).

The first two rules for Dt define an exception: If statement t_k, t_k', or t_n is empty, the last statement of the result will be empty. For example, ()+() evaluates to () and not (+), since the first rule applies with f = λ.

5.3.4 Reduction.

D_{16}: (m,s list pr nil,x) → (m,s nil ,Reduce(list pr) x)

D_{17}: (m,s list_op nil,x) → (m,s nil ,Reduce(list op) x)

The function Reduce constructs a subexpression consisting of list components separated by the given operator. The next operator must be a delimiter; otherwise it will be taken as a prefix operator and pushed onto the stack before rule D_{15} or D_{16} can be applied. For example,

$$((5,6,7,)+) \rightarrow ((5+6+7)) \quad \overset{*}{\rightarrow} \quad 18.$$

Note that both distribution and reduction apply to matrices of any order. For example, to multiply the corresponding elements of two 3-dimensional matrices A and B and sum the components of the result matrix, we write

$$(((A*B+)+).$$

Reduce = Rd*, where Rd is the function defined below. Reduction is carried out by the repeated application of the second rule for Rd.

$$\begin{aligned} \text{Rd:}\quad & \underline{(}t\underline{,}\qquad\underline{)}\ \text{op} \rightarrow \underline{(}t\qquad\qquad\underline{)} \\ & \underline{(}t\underline{,}\ \text{vs}\underline{)}\ \text{op} \rightarrow \underline{(}t\ \text{op}\ \text{vs}\underline{)}\ \text{op} \\ & \underline{(}\qquad\quad\underline{)}\ \text{op} \rightarrow \underline{(}\qquad\qquad\underline{)} \end{aligned}$$

5.3.5 Records.

D_{18}: $(m,s\ \underline{(}fs\underline{)}\ \underline{\&}\ \underline{(}fsl\underline{)}\ ,x) \rightarrow (m,s\ \underline{(}fs\ fsl\underline{)}\ ,x)$

D_{19}: $(m,s\ \ \underline{dom!}\ \ rec\ \ ,x) \rightarrow (m,s\ Dom(rec)\ \ ,x)$

D_{20}: $(m,s\ \ \underline{rge!}\ \ rec\ \ ,x) \rightarrow (m,s\ Rge(rec)\ \ ,x)$

Operators on records include & (concatenate) and the built-in functions dom (domain) and rge (range) which return respectively the list of field names in a record and the list of values it contains.

Ex_1: $(A{:}5,B{:}7,)\ \&\ (C{:}9,) \quad \overset{*}{\rightarrow} \quad (A{:}5,B{:}7,C{:}9,)$

Ex_2: $dom(A{:}5,B{:}7,)\ \&\ rge(C{:}9,) \quad \overset{*}{\rightarrow} \quad (A,B,)\ \&\ (9,)$

5.3.6 Record Subexpressions.

D_{21}: $(m,s\ w\ \underline{(}\ fs\ name\ \ ,\underline{:}y) \rightarrow (m,s\ w\ \underline{(}\ fs\ name\ \ \underline{:}nil,y)$

D_{22}: $(m,s\ w\ \underline{(}\ fs\ field,\underline{,}y) \rightarrow (m,s\ w\ \underline{(}\ fs\ field\underline{,}nil,y)$

D_{23}: $(m,s\ w\ \underline{(}\ fs\ nil\ \ ,\underline{)}y) \rightarrow (m,s\ w\ ,\ \underline{(}fs\underline{)}\ \ \ \ y)$

D_{24}: $(m,s\ w\ \underline{(}\ fs\ field,\underline{)}y) \rightarrow (m,s\ w\ ,\ \underline{(}fs\ field,\underline{)}\ \ y)$

An expression of the form $(n_1\colon e_1, \ldots, n_k\colon e_k,)$ where the n_i are field-names and the e_i are expressions with value v_i yields a record of the form $(n_1\colon v_1, \ldots, n_k\colon v_k)$ upon evaluation. As with lists, the final comma can be omitted.

Ex_1: (A:5+0,B:3*2) $\overset{*}{\rightarrow}$ (A:5,B:6,)

Ex_2: (A,B).: (5,6) $\overset{*}{\rightarrow}$ (A:5,B:6,)

5.4 Variables.

5.4.1 Assignment.

```
V1:  (m         ,s con ev var,x) → (Find(m,var),s con cv var  , x)

V2:  (m         ,s nil ev var,x) → (Find(m,var),s nil ev var  , x)

V3:  (y→con yl,s conl → var,x) → (y   conl yl,s          conl, x)

V4:  (y→v   yl,s val !  var,x) → (y   v    yl,s          v   , x)
```

The operations of assignment ($\underline{\rightarrow}$) and retrieval are carried out in two steps: first, the referenced value is located; then it is either replaced by a constant (V_3) or retrieved (V_4).

The value of Find(m,var) is the memory string m with the cursor $\underline{\rightarrow}$ inserted just before the value referenced by the variable var. For example,

```
if             m    = '(A:5,B:(3,4),)'

then Find(m,B)      = '(A:5,B:→(3,4,),)'

and  Find(m,B.2)    = '(A:5,B:(3,→4,),)'

and  Find(m,C.1)    = '(A:5,B:(3,4,),→)'
```

Records are searched from left to right until a matching name is found or failure occurs, as in the last example. Lists are indexed starting with 1, so B.2 references the second item in list B. Lists and records may be nested to any depth, and variables may have any number of field names and indices as subscripts. Multiple assignment is possible, as shown below.

```
((B:0,C:5,D:6,) , ⊢ , val!B→C→D⊣)   *→   ((B:0,C:0,D:0), ⊢0, ⊣).
```

5.4.2 Input/Output.

```
V5:  (y→ (v, vs) yl,s read! var,x) → (y(vs)       yl,s v      ,x)

V6:  (y→ (vs v,) yl,s  pop! var,x) → (y(vs)       yl,s v      ,x)

V7:  (y→ (vs    ) yl,s con→> var,x) → (y(vs con,) yl,s

                                      var.Len((vs con,)),x)
```

The built-in functions read and pop and the operator $\underline{\rightarrow>}$ (write or push) are defined on variables which reference a list. read and pop remove the first and last element of the list respectively and return it as their value; $\underline{\rightarrow>}$ appends its left operand to the end of the list and returns a subscripted variable that references the appended element. For example, if file F1 is initially bound to the null list $\underline{()}$ we have

$$((A:5,B:6,) \rightarrow> F1,val(F1.1),read(F1),) \overset{*}{\rightarrow} (F1.1,(A:5,B:6,),(A:5,B:6,),)$$

5.4.3 Dereferencing.

$$V_8\text{:}\quad (y \underline{\rightarrow} \text{ con } yl,s \quad \underline{ref}\ \underline{!}\ var,x) \rightarrow (y \text{ con } yl,s \qquad var,x)$$

$$V_9\text{:}\quad (y \underline{\rightarrow} v \quad yl,s \quad w \text{ ev } var,x) \rightarrow (y\ v \quad yl,s\ w \text{ ev } v,x)$$

$$V_{10}\text{:}\quad (m \qquad ,s \text{ deref } \underline{!} \text{ con},x) \rightarrow (m \qquad ,s \qquad \text{con},x)$$

The function ref returns a reference to a constant when applied to a variable var. If var directly references a constant, var itself is returned (V_8). If var directly references another variable, the variable will be retrieved (V_9) and dereferenced in turn. Rule V_9 ensures that if the chain of references is not dangling or circular, $\underline{\rightarrow}$ and $\underline{\rightarrow>}$ will coerce their right operands to a reference to a constant, and other evaluators will obtain a constant. In particular, the function con returns the constant indirectly referenced by its argument (V_{10}).

The last rule permits the functions <u>val</u>, <u>ref</u>, and <u>con</u> in the class Deref to be applied componentwise to parameter lists containing constants as well as variables. For example, if m = '(A:5,B:A,C:B,D:C,)' then by rules D_{14}, V_4, V_8, and V_{10}

$$\text{val.!(5,A,B,C,D,)} \overset{*}{\rightarrow} \text{(5,5,A,B,C,)}$$

$$\text{ref.!(5,A,B,C,D,)} \overset{*}{\rightarrow} \text{(5,A,A,A,A,)}$$

$$\text{con.!(5,A,B,C,D,)} \overset{*}{\rightarrow} \text{(5,5,5,5,5,)}$$

5.4.4 <u>Subscripts</u>.

```
V11: (m           ,s var ev w  ,x) → (Find(m,var),s nil (var,ev  w) ,x)
V12: (y→list yl,s var,!varl x) → (y list yl,s var,con! varl      x)
V13: (y→list yl,s var,!i    x) → (y list yl,s var.i            ,x)
V14: (y→rec  yl,s var,!name x) → (y rec  yl,s var.name         ,x)
V15: (y→stru yl,s var,!list x) → (y stru yl,s nil,Dist(var!list) x)
V16: (y→v    yl,s var,ev w  x) → (y v    yl,s v  ,ev w          x)
```

The operator <u>!</u> (subscript) appends an index to a list-valued variable (V_{13}) and a subfield name to a record-valued variable (V_{14}). Variables used to index a list are coerced to their values (V_{12}). Variables distribute over lists of subscripts (V_{15}).

Evaluators other than $\underline{!}$ dereference their left-hand operand and then their right-hand operand, if both are variables. Names of built-in functions and types are exceptions: since they are also constants, rules V_1 through V_9 will be applied first. If the right-hand variable dereferences to a constant for which the function is not defined, the function name will be dereferenced.

For example, suppose that memory contains an array A and a record Employee as follows:

```
m = '(A:(1,2,3,),K:2,
     Employee:  (Name: (First: ''Joe'',Last: ''Smith'',),
                 Skills: (5,6,8,9,),),)'
```

Then the examples below are valid:

$A(K) \overset{*}{\rightarrow} A!K \overset{*}{\rightarrow} A!2 \overset{*}{\rightarrow} A.2$

$\text{Employee Name (First)} \overset{*}{\rightarrow} \text{Employee.Name.First}$

$A(A) \overset{*}{\rightarrow} A!(1,2,3,) \overset{*}{\rightarrow} (A.1,A.2,A.3,)$

$\text{Employee(Name,Skills)} \overset{*}{\rightarrow} \text{(Employee.Name,Employee.Skills,)}$

$\text{len!A} \overset{*}{\rightarrow} \text{len!(1,2,3,)} \overset{*}{\rightarrow} 3$

$A{\rightarrow}A \overset{*}{\rightarrow} (1,2,3,){\rightarrow}A$

5.4.5 Environment.

```
V17:  (y→) ,s ref$ sl env,x) →  (y)  ,s ref$ sl ref     ,x)

V18:  (y→) ,s ref$ sl var,x) →  (y)  ,s ref$ sl ref.var,x)

V19:  (y→yl,s         var,x) →  (y yl,s null            ,x)
```

Variables without a corresponding value default to null (V_{19}). There is an exception: If the variable is not global (i.e. it fails to match a field-name of the memory record), if memory contains the global variable r bound to a list of records (local environments), and if the execution stack contains a reference followed by $, then the reference is appended to the variable (V_{18}) so that Find will try to resolve the variable in the local environment. If Find fails a second time the exception no longer holds, since r is global, and null is returned. The special variable env returns a reference to the current local environment (V_{17}).

For example, in the memory m below, sysin, sysout, and r are global variables, while job, rate, and code are local variables.

```
m = '(sysin:(''12 10'',''30 50'',),
      sysout:(''TOTAL=56 92'',),
      r:((job:5,rate:2.50,code:300      ,),
         (job:6,rate:3.10,code:r.1.code,),
         (job:7,rate:4.25,code:r.2.code,),),)'
```

An attempt to dereference the variable A leads to the following computation, where y and yl are strings such that m = y) = yl),),):

```
(m          ,⊢r.3$+A       ,⊣)
(y→)        ,⊢r.3$+A       ,⊣)
(y )        ,⊢r.3$+r.3.A   ,⊣)
(yl→),),)   ,⊢r.3$+r.3.A   ,⊣)
(m          ,⊢r.3$+%       ,⊣)
```

5.4.6 Blocks.

```
V_20:  (m,sd        ref,$ y) → (m,sd      ref $  nil ,  y)
V_21:  (m,sd        var,$ y) → (m,sd    - ref !  var ,$ y)
V_22:  (m,sd        rec,$ y) → (m,sd      rec →>  r  ,$ y)
V_23:  (m,sd        wl  ,; y) → (m,sd            nil ,  y)
V_24:  (m,s ref$    w   ,, y) → (m,s               w ,, y)
V_25:  (m,s ref$    w   ,) y) → (m,s               w ,) y)
```

where sd ∈ (Stack ((| , | Delimiter)).

A block is a statement of the form e $ t where e is a declaration (i.e. an expression whose value is a record or a record-valued variable) and t is a statement called the scope of the declaration. When the value of e is a reference, it is pushed onto the stack (V_{20}). Variables are resolved first (V_{21}), and records are pushed onto the environment stack (V_{22}).

Statement t is evaluated in the new environment. When t is terminated by a , or), the environment pointer is deleted and the previous environment restored (V_{24},V_{25}). The value of a compound statement of the form e_1; e_2; ... e_k is the value of e_k (V_{23}). Note that compound statements presuppose the existence of variables: if the first k-1 expressions didn't have side-effects there would be no reason to evaluate them.

The block below contains the declaration of a single variable, Pay, whose value is a record with two fields. Note the second use of $ to simplify record processing.

```
(Pay:(Regular:(Hrs:8,Rate:4.53,Amount:0),
      Overtime:(Hrs:8,Rate:6.75,Amount:0)))$
 Pay.Regular.Hrs * Pay.Regular.Rate → Pay.Regular.Amount;
        (Pay.Overtime$ Hrs * Rate → Amount)
```

The relationship between program and data structure is more obvious if we write

```
(Pay$(Regular $ Hrs * Rate → Amount   ,
      Overtime $ Hrs * Rate → Amount))
```

In the example below, the variable A is declared local to three successive inner blocks.

```
(A:5,B:0) $ ((A:7)                                $A+B→A→B       );
            ((A:8) & env                          $A+B→A→B       );
            ((A:9) & (dom(env).:ref.!dom(env))$A+B→A→B;pop(r));
```

The first block is closed, i.e the environment of the enclosing block is inaccessible. A+B is undefined and A will be assigned the value null. In the second block, A+B is evaluated in an environment formed by concatenating a copy of the enclosing environment to the local activation record. The local variable A will be set to 8 and the enclosing environment will be unaffected. This permits backtracking behavior as in the PLANNER language.

The third subexpression resembles an Algol 60 block: The local variable B is bound to a reference to the nonlocal B, which will be reset to 9. Since the second occurrence of A in the local activation record is inaccessible, the nonlocal A will be unchanged. As in Algol, the activation record is deleted when the block is exited.

5.5 Procedures.

5.5.1 Execution.

P_1: `(m,s     nil ![f]   ,x) → (m,s nil ,(f)            x)`

P_2: `(m,s [name f ]!con  ,x) → (m,s nil ,((name:con)   f)   x)`

P_3: `(m,s [list f ]!list1,x) → (m,s nil ,((list.:ref.!list)f) x)`

A procedure is a form enclosed in square brackets. The prefix operator ! (execute) causes a procedure to be executed by replacing the brackets with parentheses and placing the resulting subexpression in the input buffer for evaluation (P_1). The infix operator ! (apply) applies a procedure to a constant (P_2) or a parameter list (P_3). In the second case, function ref coerces variable parameters to references in the environment of execution; then the operator .: constructs an activation record which binds formal parameters to (references to) actual parameters. This is illustrated in the example below, where each procedure is used to increment A by 5.

```
P: [A+5 → A]         ...     !P
F: [X$X+5]           ...     F(A) → A
G: [(X,Y)$X+5 → Y] ...     G(A,A)
H: [R$R$X+5 → Y]     ...     H(X:val(A),Y:ref(A))
```

5.5.2 Environment.

P_4: `(m,s rec ! [ e $ f ],x) → (m,s [ e & rec $ f ],x)`

Rule P_4 permits part of the local environment of a procedure to be computed at definition time and then inserted into the procedure. Its use is illustrated by the definition of procedures I and J below:

```
(A:50,B:60)$
(A:5,B:6,F:[X & (A:10)     $ X + A → A],
          G:[X & env       $ X + A → A],
          H:[X & (A:ref(A)) $ X + A → A],
          I: env          [X $ X + A → A],
          J:(A:ref(B))    [X $ X + A → A])$ F(5); H(5); I(5); J(5);
```

F(5) = 15 since A is a local variable initialized to 10. G(5) = 10, but since G contains a <u>copy</u> of its environment of execution, it has no side-effects. H(5) returns 10 and modifies its environment of execution since the local variable A is bound to a reference. I(5) returns 55 and has no side-effects, while J(5) resets B in the environment of definition to 65.

The procedure Common defined below accepts records and record-valued references as input, and returns a record of references.

```
Common:[(R,)$R$[dom(env)](dom(env))]
```

Appending the value of Common(R) to a procedure's environment permits the procedure to reference (access and modify) the components of a record R by name. Common can be used to permit shared COMMON blocks, as in Fortran, or to support <u>own</u> variables and recursion as in the example below.

```
Factorial:  Common(Uses:0)  [N&Common(env,)$
                             Uses + 1 → Uses;
                             N = 0 => 1,
                             N * Factorial(N-1)]
```

The first instance of Common pushes the record (Uses:0) onto the environment stack and returns a record of the form (Uses:r.n.Uses), which is appended to the local environment of procedure Factorial. The second instance of Common produces a record of reference to the enclosing environment when Factorial is executed. A field of the form Factorial:r.n.Factorial must be included, so the local environment of Factorial will include references to both the (non-local) 'own' variable Uses, which is used to count the number of times Factorial is executed, and Factorial itself.

5.5.3 Coercion.

P_5: (m,s $\underline{[}$f$\underline{]}$ pr $\underline{[}$f1$\underline{]}$,x) → (m,s nil , Dist($\underline{(}$f$\underline{)}$ pr $\underline{(}$f1$\underline{)}$) x)

P_6: (m,s $\underline{[}$f$\underline{]}$ pr con ,x) → (m,s nil , Dist($\underline{(}$f$\underline{)}$ pr con) x)

P_7: (m,s con pr $\underline{[}$f$\underline{]}$,x) → (m,s nil , Dist(con pr $\underline{(}$f$\underline{)}$) x)

P_8: (m,s nil pr $\underline{[}$f$\underline{]}$,x) → (m,s nil , Dist(nil pr $\underline{(}$f$\underline{)}$) x)

The rules above coerce a procedure to a value by executing it. When the procedure consists of a list of statements, the operator and other operand are distributed over it as in the example below.

$$9 + [6 \rightarrow A, 7 \rightarrow B] \quad \overset{*}{\rightarrow} \quad (9+6 \rightarrow A, 9+7 \rightarrow B)$$

Note that if parentheses were used in place of square brackets, the assignment would be carried out before the addition rather than after.

The coercion rules above also permit non-procedural language features to be modeled. For example, the first three expressions below can be executed in any order without changing the meaning (output) of the procedure.

```
[ [X*X] → A;
  [Y*Z] → B;
  [A-B] → C;
    0.→ (X,Y,Z);!C →> Output;
   25.→ (X,Y,Z);!C →> Output ]
```

5.6 Control Structures.

The conditional and loop operators $\underline{=>}$ and $\underline{@}$ are defined only for states of the form (m,sd w,x), where sd is a stack belonging to the syntactic class Stack ($\underline{(}$ | $\underline{\bot}$ | Delimiter).

5.6.1 Conditionals.

C_1: (m,sd true ,$\underline{=>}$t f$\underline{)}$ y) $\rightarrow$ (m,sd nil, t $\underline{)}$ y)

C_2: (m,sd false,$\underline{=>}$t f$\underline{)}$ y) $\rightarrow$ (m,sd nil, f $\underline{)}$ y)

A subexpression of the form (e_1 => t_1, e_2 => t_2, ...) has the expected meaning: Successive expressions e_i are evaluated until the value true is returned; then the corresponding statement t_i is executed and the subexpression is exited. Some examples are given below along with their syntactically sugared equivalents.

```
      (X > Y => 25 → Y)  ≡  if X > Y then 25 → Y fi
       (P(x) => A   , B)  ≡  if P(x) then A else B fi
   (P(x) => Q(x) => 1,0)  ≡  (P(x) and Q(x))
  (P(x) => 1,Q(x) => 1,0)  ≡  (P(x) or Q(x))
```

In the last two examples Q(x) is executed only when P(x) fails to determine the value of the Boolean expression.

5.6.2 Loops.

C_3: (m,sd nil, $\underline{@}$ t $\underline{)}$ y) → (m,sd nil, t $\underline{,@}$ t $\underline{)}$ y)

A subexpression of the form (@t) causes statement t to be executed repeatedly until the loop is exited (or $\underline{\vdash}$ is entered from the terminal). Two possible forms are illustrated below together with their syntactically sugared equivalents.

```
  (@ el  ; e2 =>)  ≡  do el until e2 od
(@ el  ; e2 => e3)  ≡  do el until e2 returning e3 od
```

The program fragment below copies non-zero integers from an input to an output file, skipping zeroes and exiting when the file is exhausted or when a non-integer value is encountered. Assume exit = 1 and repeat = 0.

```
(@(len(input) = 0 => exit,
     read(input) → x;
    ¬int(x)  => exit,
      x = 0  => repeat,
     x →> output;repeat)=>)
```

By enclosing conditionals in square brackets and applying the coercion rules of section 5.5.3 we obtain the forms of labelled <u>case</u> expressions shown below.

```
i = [1 => P(x), 2 => Q(x), i => Default(x)]

x = [Apples => Case1(), Oranges => Case2(), Fruit => 0]

i = [1 =>A, 3 => B, (2,4,9)∨ => C, (5..8)∨ => D]

x < [0 => error, 10 => rate1, 20 => rate2, 0; error]
```

5.6.3 <u>Indexed Loops</u>.

C_4: $(m,sd\ \underline{0,@}\ t\ f\ \underline{)}\ y) \rightarrow (m,sd\ nil,\qquad f\ \underline{)}\ y)$

C_5: $(m,sd\ i,\underline{@}\ t\ f\ \underline{)}\ y) \rightarrow (m,sd\ nil,\ t\ \underline{,i-1@}\ t\ f\ \underline{)}\ y)$

A subexpression of the form (f1;i@t f2) is evaluated by executing f1, executing statement t i times (or until the loop is exited), and then executing f2. The four examples below perform the following actions: copying 10 records from file A to file B; selecting a subset of a list of names using a bit map; creating a vector; creating a 2 by 3 matrix.

(10@ read(A) $\twoheadrightarrow$ B;) $\overset{*}{\rightarrow}$ ()

(1,0,1).@(FRED,JOE,MIKE) $\overset{*}{\rightarrow}$ (FRED,MIKE,)

(1,2,3@0,2@4) $\overset{*}{\rightarrow}$ (1,2,0,0,0,4,4,)

(2@(3@0)) $\overset{*}{\rightarrow}$ ((0,0,0,),
(0,0,0,),)

Note that the value of the first expression is the empty list because the statement is terminated with a semicolon. Note also that dimensions are given in the same order as subscripts: When A = (2@(3@0)), A.2.3 is defined but A.3.2 is not.

5.6.4 For Loops.

C_6: (m,sd $\underline{\texttt{''}}$,@ t f$\underline{)}$ y) → (m,sd nil, f$\underline{)}$ y)

C_7: (m,sd str,@ t f$\underline{)}$ y) → (m,sd $\underline{\texttt{hd!}}$ str,t $\underline{\texttt{,tl!}}$ str $\underline{@}$ t f$\underline{)}$ y)

C_8: (m,sd $\underline{\texttt{()}}$,@ t f$\underline{)}$ y) → (m,sd nil, f$\underline{)}$ y)

C_9: (m,sd list,@ t f$\underline{)}$ y) → (m,sd $\underline{\texttt{hd!}}$list, $\underline{\texttt{,tl!}}$list @ t f$\underline{)}$ y)

A subexpression of the form (list@ → x; e) evaluates expression e for every value x in list. Loops can be nested as in the second example below.

```
((1,3,5,2)@ → I; A(I) + B(I) → C(I))
(1..5@ → I; 4..20@ → J; e(I,J))
```

Lists can be constructed from character strings and matrices as follows:

$$(\texttt{'ABC'@}) \rightarrow (\texttt{'A','BC'@}) \overset{*}{\rightarrow} (\texttt{'A','B','C',})$$

$$(((0,0),(1,1))@@) \overset{*}{\rightarrow} ((0,0)@,(1,1)@) \overset{*}{\rightarrow} (0,0,1,1,)$$

Using the distributive property of the subscript operator, we can reverse the rows and columns of a 9 x 8 x 7 matrix M by componentwise assignment:

```
(M(9..1)@(8..1)@(1..7)).→ (M(1..9)@(1..8)@(1..7))
```

5.6.5 Coercion.

C_{10}: (m,sd var,=> y) → (m,sd val!var, => y)

C_{11}: (m,sd var,@ y) → (m,sd val!var, @ y)

Operators => and @ are the only delimiters which coerce their operands. Since variables are dereferenced, expressions like the one below are meaningful.

```
(P => N@O)
```

The operators .=> and .@ also dereference variable operands. In the example below, the expressions on the left have the meaning suggested by the syntactically sugared equivalent expressions on the right when L is a list.

```
(L > 0).=> L  ≡  first x in L such that x > 0 end
(L > 0).@  L  ≡  all x in L such that x > 0 end
```

5.7 Self-Extension.

S_1: $(m,s\ w\ ev\ wl, \quad x) \rightarrow (m,s\ \underline{ext!(}\ w\ \underline{,'}\ ev\ \underline{',}\ wl\underline{)}\ ,x)$

S_2: $(m,s\ \ \underline{(}\ \ f\ ,\ \underline{)}\ \ y) \rightarrow (m,s\ \underline{err![}f\underline{]} \qquad ,y)$

S_3: $(m,s \qquad w\ ,\ op\ y) \rightarrow (m,s\ \underline{err![}\ w\ \underline{,'}\ op\ \underline{']} \qquad ,y)$

Since rules S_1, S_2 and S_3 are the last rules of the definition they provide a default action for states to which no other rule applies. Let us assume that procedures ext and err always return a value. (This will be the case if they are undefined, by rule R_1 in section 5.2.1.) Then we can prove

Theorem: Trivial expressions (those containing only separators) reduce to nil. Non-trivial expressions whose evaluation terminates always reduce to a value.

Proof: Rules S_1 and E_8 together ensure that the theorem holds for expressions whose subexpressions yield a value. Rules S_2, S_3, and E_5 ensure that every subexpression whose evaluation terminates yields a value. Q.E.D.

Subexpressions or portions of subexpressions which cannot otherwise be evaluated are encapsulated in square brackets and passed to an error procedure for correction or display (S_2,S_3). An erroneous delimiter or $\underline{,}$ is deleted before being pushed onto the stack so that an interactive user can enter a correction.

The language extension procedure ext is applied to binary expressions which are otherwise undefined. Since ext obeys the same scope rules as other user-defined procedures, extensions can be made local to particular blocks. For example, the operator * will be extended so that

$$3 * \text{'A'} \equiv \text{'A'} * 3 \equiv \text{'AAA'}$$

for all expressions whose environment includes

```
    ext: [(w,op,wl)$
         (w,op,wl)=[(int,'*',str)^ => (w@wl).&,
                   (str,'*',int)^ => (wl@w).&]],
```

6.0 SIBYL Pragmatics.

The meaning assigned to a program by a formal language specification is in general incomplete in that it fails to describe the interaction of the program with the peripheral devices and software of the particular installation on which the program is run. For example, the expression '+TOTAL=10' $\twoheadrightarrow$ P acquires additional meaning when output file P is attached to a printer that uses $\underline{+}$ as a carriage control.

The notation introduced in section 3.6 permits the definition of computer installations as networks of concurrently executing hardware and software modules and provides a means for defining parallel and non-deterministic language features. To see this, consider the hardware configuration diagrammed in figure 5. It consists of an operator's console, a tape unit, a printer, and three processors, all sharing a common memory. Assuming that the string automata Console, Tape, and Printer have been defined, the configuration is completely specified by

$\mathrm{Inst}(R) = I_1 + I_2 + I_3 + C + T + P$ where

$I_i = \mathrm{Processor}(\mathrm{Mem},\mathrm{Stack}_i, \mathrm{Input}_i)$,

$C = \mathrm{Console}(\mathrm{Mem},\mathrm{Display},\mathrm{Input}_1,\mathrm{Keys})$,

$T = \mathrm{Tape}(\mathrm{Mem},\mathrm{Reel})$,

$P = \mathrm{Printer}(\mathrm{Mem},\mathrm{Output})$,

$R = (\mathrm{Mem},\mathrm{Display},\mathrm{Keys},\mathrm{Reel},\mathrm{Output})\ \&\ \mathrm{Stack}_{1..3}\ \&\ \mathrm{Input}_{1..3}$,

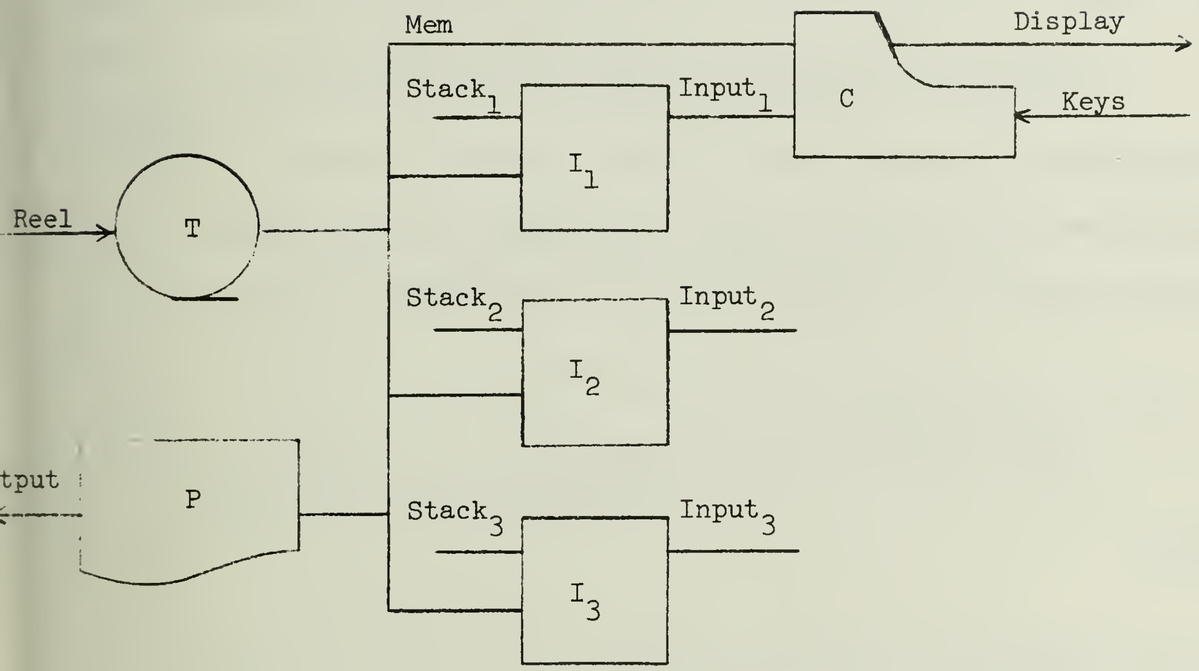


Figure 5: Multiprocessor System.

and Processor is defined by

$$\text{Processor} = E_{1..7} \,\&\, R_{1..29} \,\&\, D_{1..24} \,\&\, V \,\&\, P_{1..8} \,\&\, C_{1..11} \,\&\, S_{1..3}$$

$$V = (V_{3..9} \,\&\, V_{12..19}) \circ (V_1, V_2, V_{11}) \,\&\, V_{10} \,\&\, V_{20..25}$$

The function composition operator o is used to combine the two steps of a memory access into one indivisible operation. This eliminates intermediate states of the form (y$\underline{\rightarrow}$y1,stack,input) and permits multiple processors to share the same memory register.

6.1 Parallel Tasks.

If the memory, stack, and input registers are initialized so that

```
mem = (r:(),Tasks:(),C:(),T:(),P:(),)
stack  = ⊢
     i
input  = (@!read(Tasks);)
     i
```

programs entered from the console can initiate parallel tasks by placing them on the Tasks queue for subsequent execution by processors I_1 and I_2. An example is

```
⊢((Producer: [e1] ,Consumer: [e2 ],)$
   Producer →> Tasks; Consumer →> Tasks)⊣
```

Let us suppose that Producer is a procedure that reads records from the tape file T, processes them, and passes them onto Consumer via a queue Q. Consumer processes records from Q and outputs them to the printer via file P. The program that spawns both tasks must supply them with Q and a procedure Read that waits until a file contains an entry before returning to the calling program.

```
((Read: [X & (Y:0)$@ ¬((read(X)→Y)=null) => val(Y)],
  Q: ()                                        ,)$
 (Producer: env [(R:0)$@Read(T)→R;e1;R→>Q;R=()=>],
  Consumer: env [(R:0)$@Read(Q)→R;e2;R→>P,R=()=>],)$
 (Producer,Consumer)@→>Tasks)
```

In the example above, both tasks continue to execute after the main program terminates, and remain active until the empty record () (used here as an end-of-file marker) is read and passed on.

6.2 Semaphores and Coroutines.

Queues like Q in the previous example may be used as semaphores. In the example below, semaphore S ensures that only one task at a time is in its critical section, while C1 and C2 are used to simulate coroutine linkage.

```
(Pause: [X$@¬read(X)=null)=>], S:(1,),C1:(),C2:(),
 Producer: [e1;Pause(S); critical section1; 1→>S; e2],
 Consumer: [e3,Pause(S); critical section2; 1→>S; e4],
 Coroutine1: [e1; 1→>C2; Pause(C1); e3; 1→>C2],
 Coroutine2: [Pause(C2); e2; 1→>C1; Pause(C2); e4])$
 (Produces, Consumes, Coroutine 1, Coroutine 2) @→>Tasks;
```

7.0 Summary.

The semantics of both the hardware and software components of a computing system can be precisely described by a formal notation based on lists of transition rules over n-tuples of character strings. The formalism used in this paper seems especially suitable for describing artificial languages to their users, for the following reasons:

(1) The notation is simple and requires little mathematical sophistication on the part of its users.

(2) Definitions are operational. They provide a straightforward procedure for evaluating expressions in the defined language.

(3) Every state of a computation has a unique concrete representation as a tuple of character strings.

(4) Definitions are modular. Transition rules can be reordered and combined in various ways to facilitate learning by different classes of users, or to provide alternate views of the same language as an aid to understanding.

These same characteristics make the formalism attractive as a language design tool. Its simplicity makes it amenable to mathematical definition and analysis, its operational nature permits the automatic synthesis of interpreters for example programs, and its modularity supports a building-block approach to language design.

REFERENCES

Aho, A. V. and J. D. Ullman [1972]
The Theory of Parsing, Translation and Compiling.
Prentice-Hall (Englewood Cliffs, N.J.).

Backus, J. W. [1959]
"The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference"
Proc. International Conf. on Information Processing, UNESCØ, pp. 125-132.

Dennis, J.B. [1972]
"On the Design and Specification of a Common Base Language"
MAC TR-101 MIT Project MAC (Cambridge).

Feyock, Stefan [1975]
"Toward an Implementation of the Vienna Definition Language"
Proc. 1975 International Conference on ALGOL 68.
Oklahoma State Univ. (Stillwater), pp. 370-384.

Garwick, J. V. [1966]
"The Definition of Programming Languages by their Compilers"
Formal Language Description Languages for Computer Programming (Proc. IFIP Working Conf. 1964) (Steel, T. B., Ed.).
North-Holland Publ. Co. (Amsterdam) pp. 266-294.

Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright [1975]
Initial Algebra Semantics and Continuous Algebras
Research Rept. RC 5243, Thomas J. Watson Research Center, (Yorktown Heights, New York).

Herriot, R. G. [1971]
The Definition of the Control and Environment Structure of Programming Languages. (Ph.D. Thesis) Univ. of Wisconsin (Madison).

Hoare, C. A. R. [1969]
"An Axiomatic Basis for Computer Programming" Comm. ACM 12:10, pp. 576-580.

Hoare, C. A. R., and P. E. Lauer [1973]
Consistent and Complementary Formal Theories of the Semantics of Programming Languages. Tech. Rept. 44, Univ. of Newcastle-upon-Tyne (Newcastle-upon-Tyne).

Irons, E. T. [1970]
"Experience with an Extensible Language" Comm. ACM 13:1 pp. 31-40.

Kampen, G. R. [1973]
SIBYL: A Formally Defined Interactive Programming System Containing an Extensible Block-Structured Language. (Ph.D. Thesis) Tech. Rept. #73-06-16, Computer Science Group, University of Washington (Seattle).

Kampen, G. R. and J. L. Baer [1975]
"The Formal Definition of Semantics by String Automata"
Computer Languages V. 1. Bergman Press pp. 121-138.

Landin, P. J. [1965]
"A Correspondence between ALGOL 60 and Church's Lambda Notation"
Comm. ACM 8:2,3, pp. 89-101, 158-165.

Lucas, P., P. Lauer, and H. Stigleitner [1970]
Method and Notation for the Formal Definition of Programming Languages.
Tech. Rept. TR 25.087, IBM Laboratory (Vienna).

Lucas, P. and K. Walk [1969]
"On the Formal Description of PL/I"
Annual Review in Automatic Programming 6, Part 3. Pergamon Press
(New York) pp. 105-182.

Lukaszewicz, L. [1976]
A Semantics Definition System (A Preliminary Description).
UIUCDCS-R-76-773, Univ. of Illinois, Dept. of Comp. Sci. (Urbana).

Marcotty, M., H. F. Ledgard and G. V. Bachmann [1976]
"A Sampler of Formal Definitions"
Computing Surveys 8:2, pp. 155-276.

Scott, D., and C. Strachy [1971]
"Towards a Mathematical Semantics for Computer Languages"
Proc. Symp. on Computers and Automata, Polytechnic Institute of
Brooklyn; also Tech. Mon. PRG-6, Oxford U. Computing Lab.,
pp. 19-46.

Tennant, R. D. [1976]
"The Denotational Semantics of Programming Languages"
CACM 19:8, pp. 437-453.

US. [1965]
COBOL: Edition 1965.
Dept. of Defense, U.S. Gov't Printing Office
(Washington, D.C.).

van Wijngaarden et al. (editors) [1975]
"The Revised Report on the Algorithmic Language ALGOL 68"
Acta Informatica 5: 1-3.
Springer-Verlag (Berlin).

Wegner, P. [1971]
"Data Structure models in programming languages" SIGPLAN Not. 6:2
(Proc. Symp. on Data Structures in Programming Languages) pp. 1-54.

Wirth, N. and H. Weber [1966]
"Euler: A Generalization of ALGOL and its Formal Definition"
CACM 9:1, pp. 13-27, CACM 9:2, pp. 89-99.

LIOGRAPHIC DATA HET	1. Report No. UIUCDCS-R-77-852	2.	3. Recipient's Accession No.
itle and Subtitle Formal Definition of the SIBYL Programming Languages			5. Report Date March, 1977
			6.
uthor(s) Garry Kampen			8. Performing Organization Rept. No.
erforming Organization Name and Address epartment of Computer Science niversity of Illinois rbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
ponsoring Organization Name and Address			13. Type of Report & Period Covered
			14.

upplementary Notes

bstracts

his report contains a complete formal description of an experimental programming guage, SIBYL, which achieves simplicity by generalizing a number of concepts and ictures found in other programming languages. The syntax of SIBYL is defined by a ext free grammar, and the semantics by an interpreter whose state transition func- consists of a sequence of string transformation rules. This approach to semantics vides an operational definition that is highly modular and well adapted to mechanical fication.

ecause of its generality, SIBYL can in turn be used as a language definition tool. apping constructs in other high-level languages into their SIBYL equivalents, the for repeated definitions of common control and data structures is avoided.

ey Words and Document Analysis. 17a. Descriptors

emantics
emantic metanotation
oftwares
anguages
ormal languages
rogramming languages
pecification languages

dentifiers/Open-Ended Terms

OSATI Field/Group

ailability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page UNCLASSIFIED	22. Price

TIS-35 (10-70)

USCOMM-DC 40329-P71